Sistema de Monitoreo Remoto y Control de GLP

Juan Enrique Rozo Tarache
Depto. Ingenieria de Sistemas
Pontificia Universidad Javeriana
Bogota, Colombia
juan.rozot@javeriana.edu.co

Julian Camilo Ramos Granada
Depto. Ingenieria de Sistemas
Pontificia Universidad Javeriana
Bogota, Colombia
julianc.ramos@javeriana.edu.co

Daniel Felipe Castro Moreno
Depto. Ingenieria de Sistemas
Pontificia Universidad Javeriana
Bogota, Colombia
castro-df@javeriana.edu.co

Maria Paula Rodríguez Ruiz
Depto. Ingenieria de Sistemas
Pontificia Universidad Javeriana
Bogota, Colombia
rodriguezr.mp@javeriana.edu.co

***Abstract*— Indoor air quality (IAQ) impacts health and safety, with indoor pollutants such as dust, allergens, harmful gases, and volatile organic compounds (VOCs) presenting various health risks. Liquefied Petroleum Gas (LPG), widely used for cooking, is of particular concern due to its potential for accidental leaks, which can result in explosions and fires. To mitigate this risk, the proposed solution features an Internet of Things (IoT) device designed to detect LPG and take immediate action to prevent accidents with aid of MQTT protocol and Wi-Fi connections. The system includes an LPG sensor, an automatic valve closure mechanism, and a notification system that alerts homeowners via Telegram. This approach aims to improve home safety and reduce the risk of gas-related incidents. The document discusses the system's design, implementation, and testing to ensure reliable performance in detecting gas leaks and triggering safety measures.**

Keywords*— *Gas Leaks, Indoor Air Quality (IAQ), Internet of Things (IoT), Liquefied Petroleum Gas (LPG), MQTT, Notification Systems, Sensor Technology, Wi-Fi.

I. DEFINICIÓN DEL PROBLEMA

Desde la primera entrega hasta la implementación práctica de este proyecto, la definición de la problemática relacionada con la calidad del aire en interiores no ha cambiado. A continuación, se presenta un resumen reiterando esta definición:

La calidad del aire en interiores (CAI) puede tener un impacto significativo en la salud y el bienestar de las personas [1]. Los contaminantes del aire interior, como el polvo, los alérgenos, los gases nocivos y los compuestos orgánicos volátiles (COV), pueden causar una variedad de problemas de salud, como irritación de los ojos y las vías respiratorias, problemas respiratorios, asma y enfermedades cardíacas [2].

La Organización Mundial de la Salud (OMS) define la transición de riesgos tradicionales a modernos como un cambio de perjuicios por desnutrición, polución de aires en interiores, pobres higiene y sanitización del agua; a un paradigma donde los principales riesgos se asocian al tabaquismo, falta de actividad física y mala calidad del aire [3]. En este contexto, surge la necesidad de contar con soluciones que ayuden a mitigar riesgos relacionados con la calidad del aire en interiores.

Un problema crítico es la presencia de gases peligrosos como el GLP, que se usa comúnmente para cocinar. La exposición accidental a este tipo de gases puede provocar serios accidentes, como explosiones y fuegos, y tiene implicaciones para la salud y seguridad de las personas. Además, la exposición a gases puede tener efectos nocivos para la salud. La inhalación de gas puede causar síntomas como dolores de cabeza, mareos, náuseas e incluso pérdida de conciencia o la muerte en exposición a altas concentraciones. Esto se debe a que el gas desplaza el oxígeno vital para la respiración en espacios cerrados.

La solución propuesta es un dispositivo IoT que puede detectar la presencia de GLP y tomar acciones para prevenir accidentes y mejorar la seguridad en el hogar. El sistema incluye un sensor especializado para detectar GLP, una alarma de acción inmediata ante la detección del gas, un mecanismo para cerrar automáticamente la válvula de gas, y un sistema de notificaciones para alertar al propietario a través de Telegram. Este tipo de tecnología puede contribuir a la prevención de accidentes domésticos y a la reducción de riesgos asociados con fugas de gas. Además, podría ayudar a reducir la contaminación del aire en interiores y mitigar preocupaciones relacionadas con el trastorno obsesivo-compulsivo (TOC) por seguridad.

En este documento se detallará el diseño de los dispositivos y la topología de la red. Además, se incluirá la documentación de cada componente y se sustentará con fotografías. También se proporcionará la documentación de la aplicación, que incluirá diagramas de clases y diagramas de secuencia. Finalmente, se presentarán las pruebas documentadas, describiendo cada prueba junto con su propósito y el resultado obtenido, acompañadas de evidencia como capturas de pantalla, fotos, etc.

II. DISEÑO DE DISPOSITIVOS Y TOPOLOGÍA

En esta sección se presenta el diseño explicado de los dispositivos receptor y actuador, así como la topología completa de la red.

A. Dispositivo receptor

Para el primer dispositivo se elaboraron los siguientes elementos:

Esquema:

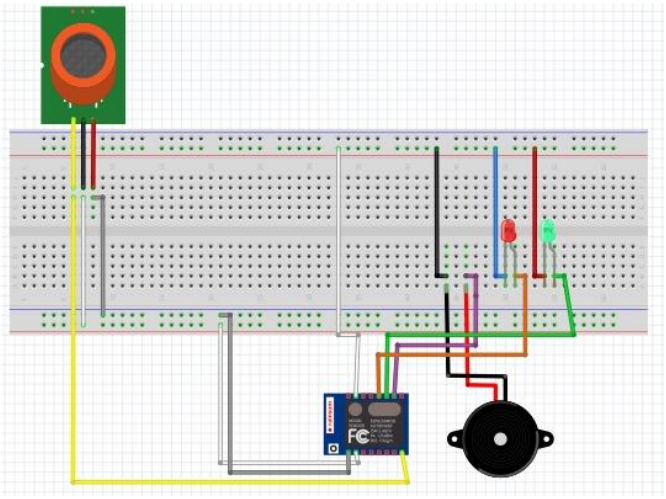

Fig. 1. Esquema del dispositivo receptor. Elaboración propia.

Componentes:

- Módulo Wifi ESP82266: [**Acá añades el texto con la descripción de cada uno, referencias, etc]**
- Buzzer:
- Placa de pruebas:
- Dos LEDs:
- Sensor de gas MQ2:
- Doce cables macho-macho (no se incluyen aquellos que unen al buzzer y el MQ2 porque estos se conectarían directamente en un diseño real):

Funcionamiento:

El circuito está diseñado para detectar la presencia de gas MQ2 en el aire. Cuando el sensor detecta gas, el LED rojo se enciende y el zumbador suena. El LED verde se enciende cuando el módulo WiFi ESP8266 está conectado a una red WiFi y no se detecta una cantidad de gas que supere el umbral establecido.

B. Dispositivo actuador

Para el primer dispositivo se elaboraron los siguientes elementos:

Esquema:

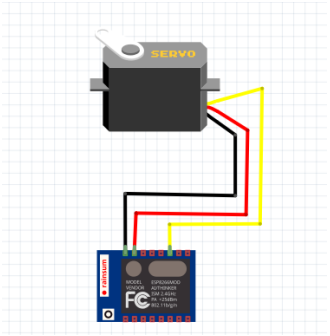

Fig. 2. Esquema del dispositivo actuador. Elaboración propia.

Componentes:

- Módulo Wifi ESP82266:
- Servomotor:
- Tres cables hembra-macho:

Funcionamiento:

El circuito está diseñado para controlar un servomotor utilizando el módulo WiFi ESP8266. El ESP8266 inicia en la posición predeterminada y gira de tal forma que simula el cierre de una llave de gas.

C. Topología

En la figura 3 se puede visualizar la topología de la red.

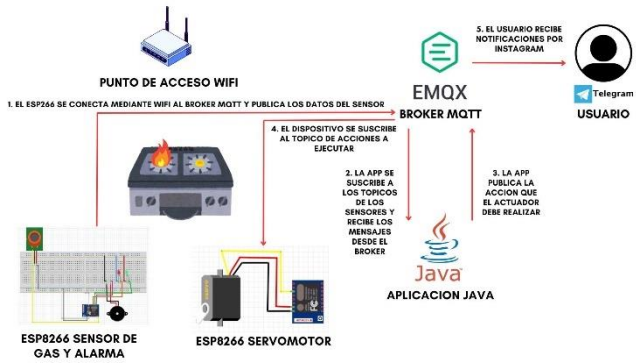

Fig. 3. Esquema del dispositivo receptor. Elaboración propia.

Como punto de partida, el primero en actuar es el ESP 266, el cual se conecta al bróker MQTT mediante WIFI para publicar los datos en el sensor. Posteriormente, la app se suscribe a los tópicos relacionados con los sensores, con el propósito de recibir los mensajes desde el bróker. En este momento, la app publica la acción que el actuador debe realizar, para que luego el dispositivo sea capaz de suscribirse al tópico de acciones a ejecutar.

Una vez realizado este proceso, el usuario recibe notificaciones constantemente vía telegram acerca del estado actual de las mediciones y/o alertas y prevenciones.

III. IMPLEMENTACIÓN DE DISPOSITIVOS

A continuación, se presentan imágenes con la elaboración física de cada dispositivo y el código para su funcionamiento. Los componentes permanecen iguales al diseño anterior.

A. Primer dispositivo

Las respectivas imágenes son las siguientes:

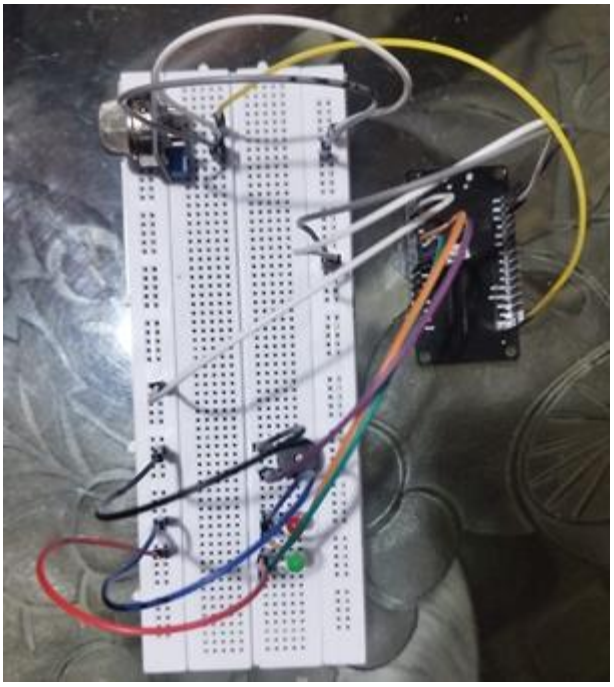

Fig. 4. Vista superior del primer dispositivo. Elaboración propia.

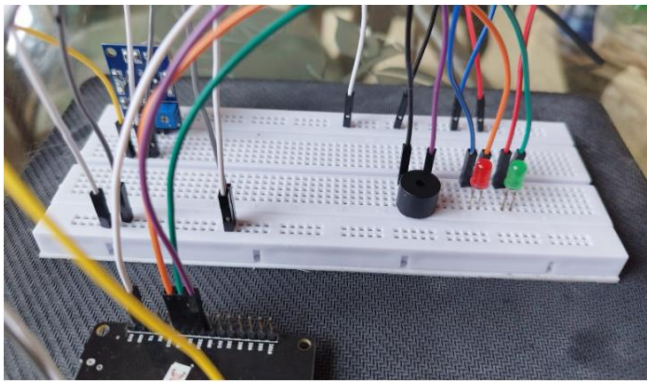

Fig. 5. Vista lateral del primer dispositivo. Elaboración propia.

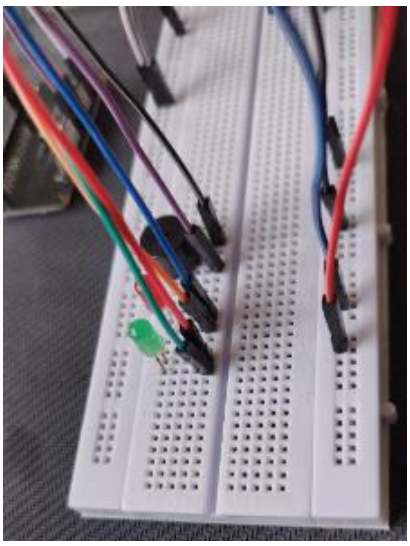

Fig. 6. Vista frontal del primer dispositivo. Elaboración propia.

El código y su análisis son los siguientes:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

#define Verde 12 // GPIO 12 (D6) LED VERDE
#define Rojo 13 // GPIO 13 (D7) LED ROJO
#define Buzzer 14 // GPIO 14 (D5) ALARMA
#define Sensor A0 // A0 SENSOR

// WiFi settings
const char *ssid = "Corporativo";
const char *password = "camicolor13";

// MQTT Broker settings
const char *mqtt_broker = "broker.emqx.io"; // EMQX broker endpoint
const char *mqtt_topic = "CANAL"; // MQTT topic
const int mqtt_port = 1883; // MQTT port (TCP)

WiFiClient espClient;
PubSubClient mqtt_client(espClient);

void connectToWiFi();
void connectToMQTTBroker();
void mqttCallback(char *topic, byte *payload, unsigned int length);
void enviarMensajeMQTT(const char *mensaje);
```

```
void setup() {
    Serial.begin(115200);
    pinMode(Verde , OUTPUT);
    pinMode(Rojo, OUTPUT);
    pinMode(Buzzer, OUTPUT);
    pinMode(Sensor, INPUT);
    connectToWiFi();
    mqtt_client.setServer(mqtt_broker, mqtt_port);
    mqtt_client.setCallback(mqttCallback);
    connectToMQTTBroker();
}

void connectToWiFi() {
    WiFi.begin(ssid, password);
    Serial.print("Conectando a WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConectado a la red WiFi");
}
```

```
void connectToMQTTBroker() {
    while (!mqtt_client.connected()) {
        String client_id = "esp8266-client-" + String(WiFi.macAddress());
        Serial.printf("Conectando al broker como %s.....\n", client_id.c_str());
        if (mqtt_client.connect(client_id.c_str())) {
            Serial.println("Conectado al broker MQTT");
            mqtt_client.subscribe(mqtt_topic);
            // Publica un mensaje al conectar exitosamente
            mqtt_client.publish(mqtt_topic, "¡Hola EMQX, soy ESP8266! ^^");
        } else {
            Serial.print("Error al conectar al broker MQTT, rc=");
            Serial.print(mqtt_client.state());
            Serial.println(" intenta de nuevo en 5 segundos");
            delay(5000);
        }
    }
}
```

```ino
void mqttCallback(char *topic, byte *payload, unsigned int length) {
    Serial.print("Mensaje recibido en el tema: ");
    Serial.println(topic);
    Serial.print("Mensaje:");
    String message;
    int i = 0;
    for (i = 0; i < length; i++) {
        message += (char) payload[i];  // Convierte *byte a string
    }
    message += '\0';

    if (message == "Alerta") {
        digitalWrite(Verde, LOW);
        digitalWrite(Rojo, HIGH);
        tone(Buzzer, 1000); // Enciende el zumbador a una frecuencia de 1000Hz
        Serial.println("La alarma está encendida");
    }
    if (message == "Solucionado" ) {
        digitalWrite(Verde, HIGH);
        digitalWrite(Rojo, LOW);
        noTone(Buzzer); // Apaga el zumbador
        Serial.println("La alarma está apagada");
    }
    Serial.println();
    Serial.println("-----------------------");
}

bool mensajeEnviado = false;
unsigned long tiempoInicio = 0;
bool tiempo = true;
```

```ino
void loop() {
    if (!mqtt_client.connected()) {
        connectToMQTTBroker();
    }
    mqtt_client.loop();

    // Lectura del sensor y publicación
    char msgBuffer[20];
    float valor_sensor = analogRead(Sensor);
    dtostrf(valor_sensor, 6, 2, msgBuffer);
    const char* valor_const_char = msgBuffer;
    enviarMensajeMQTT(valor_const_char);
    delay(5000);

    //MQTT si se detecta un nivel alto de metanol
    if (mensajeEnviado){
        float valor_metano = 165;
        dtostrf(valor_metano, 6, 2, msgBuffer);
        const char* const_metano = msgBuffer;
        enviarMensajeMQTT(const_metano);
        mensajeEnviado = false; // Marcar como enviado
    }
    //si ya paso un minuto
    if (tiempo &&(millis() - tiempoInicio >= 60000)) {
        mensajeEnviado = true; // Volver a habilitar el envío
        tiempo = false;
    }
}

void enviarMensajeMQTT(const char *mensaje) {
    mqtt_client.publish(mqtt_topic, mensaje);
}
```

Fig. 7. Código del primer dispositivo. Elaboración propia.

En el anterior código, el ESP8266 se conecta a una red WiFi y a un bróker MQTT para enviar y recibir mensajes. Para comenzar, se incluyen las librerías necesarias para manejar la conectividad WiFi y las comunicaciones MQTT. Luego, se definen las constantes para los pines GPIO que se utilizarán para controlar los LEDs, el buzzer y el sensor analógico.

A continuación, se configuran las credenciales de la red WiFi y los parámetros del bróker MQTT, como el endpoint, el tema y el puerto. También se crean los objetos necesarios para las conexiones WiFi y MQTT. Se declaran las funciones que serán utilizadas para conectarse a la red WiFi y al bróker MQTT, manejar los mensajes entrantes y enviar mensajes MQTT.

En la función setup, se inicializa la comunicación serie y se configuran los pines como entradas o salidas según corresponda con el diseño realizado en la sección anterior. Luego, el dispositivo intenta conectarse a la red WiFi utilizando las credenciales proporcionadas. Una vez conectado, se configura el bróker MQTT y se establece un callback para manejar los mensajes recibidos. Después, se intenta conectar al bróker MQTT y suscribirse al tema especificado, enviando un mensaje de bienvenida al conectar exitosamente.

La función connectToWiFi maneja la conexión a la red WiFi, intentando continuamente hasta que la conexión se establece correctamente. De manera similar, connectToMQTTBroker intenta conectar al bróker MQTT hasta que lo logra, manejando los errores y reintentando cada cinco segundos en caso de fallar.

El callback mqttCallback procesa los mensajes recibidos del bróker MQTT. Dependiendo del contenido del mensaje, se encienden o apagan los LEDs y el buzzer, proporcionando una respuesta visual y audible. Por ejemplo, si el mensaje es "Alerta", se enciende el LED rojo y el buzzer; si el mensaje es "Solucionado", se enciende el LED verde y se apaga el buzzer.

En el bucle principal (loop), se verifica si el cliente MQTT sigue conectado y, de no ser así, se reconecta al bróker. Luego, el valor del sensor se lee y se publica en el tema MQTT cada cinco segundos. Además, si ha pasado un minuto desde la última vez que se envió un mensaje específico de metano, se publica este valor y se reinicia el contador de tiempo.

Finalmente, la función enviarMensajeMQTT se encarga de publicar mensajes en el tema MQTT especificado. Esta estructura permite al dispositivo ESP8266 mantenerse conectado a la red y al bróker MQTT, enviar datos del sensor periódicamente y reaccionar a mensajes específicos recibidos, controlando los LEDs y el buzzer en respuesta a comandos.

B. Segundo dispositivo

Las respectivas imágenes son las siguientes:

Fig. 8. Vista superior del segundo dispositivo. Elaboración propia.

El código y su análisis son los siguientes:

```
#include "Servo.h"
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

#define servo_pin 13 //(D7)
#define potpin A0

Servo myservo;

// WiFi settings
const char *ssid = "Corporativo";
const char *password = "camicolor13";

// MQTT Broker settings
const char *mqtt_broker = "broker.emqx.io";  // EMQX broker endpoint
const char *mqtt_topic = "CANAL";     // MQTT topic
const int mqtt_port = 1883;  // MQTT port (TCP)

WiFiClient espClient;
PubSubClient mqtt_client(espClient);
```

```
void connectToWiFi();
void connectToMQTTBroker();
void mqttCallback(char *topic, byte *payload, unsigned int length);
void enviarMensajeMQTT(const char *mensaje);

void setup() {
  Serial.begin(115200);
  connectToWiFi();
  mqtt_client.setServer(mqtt_broker, mqtt_port);
  mqtt_client.setCallback(mqttCallback);
  connectToMQTTBroker();
  myservo.attach(servo_pin);
}

void connectToWiFi() {
    WiFi.begin(ssid, password);
    Serial.print("Conectando a WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConectado a la red WiFi");
}
```

```
void connectToMQTTBroker() {
    while (!mqtt_client.connected()) {
        String client_id = "esp8266-client-" + String(WiFi.macAddress());
        Serial.printf("Conectando al broker como %s.....\n", client_id.c_str());
        if (mqtt_client.connect(client_id.c_str())) {
            Serial.println("Conectado al broker MQTT");
            mqtt_client.subscribe(mqtt_topic);
            // Publica un mensaje al conectar exitosamente
            mqtt_client.publish(mqtt_topic, "¡Hola EMQX, soy ESP8266! ^^");
        } else {
            Serial.print("Error al conectar al broker MQTT, rc=");
            Serial.print(mqtt_client.state());
            Serial.println(" intenta de nuevo en 5 segundos");
            delay(5000);
        }
    }
}
```

```
void mqttCallback(char *topic, byte *payload, unsigned int length) {
    Serial.print("Mensaje recibido en el tema: ");
    Serial.println(topic);
    Serial.print("Mensaje:");
    String message;
    int i = 0;
    for (i = 0; i < length; i++) {
        message += (char) payload[i];  // Convierte *byte a string
    }
    message += '\0';

    if (message == "Alerta") {

      for(int ang=0; ang < 180; ang++){
        myservo.write(ang);
        delay(20);
      }
      delay(500);

      enviarMensajeMQTT("SERVOACTIVADO");

      Serial.println("El servomotor esta prendido");
    }
```

```
    if (message == "Solucionado" ) {
      Serial.println("El servomotor quedo en su lugar");
    }
    Serial.println();
    Serial.println("-----------------------");
}

void loop() {
    if (!mqtt_client.connected()) {
        connectToMQTTBroker();
    }
    mqtt_client.loop();
}

void enviarMensajeMQTT(const char *mensaje) {
    mqtt_client.publish(mqtt_topic, mensaje);
}
```

Fig. 9. Código del segundo dispositivo. Elaboración propia.

El programa comienza incluyendo las librerías necesarias para controlar el servomotor y para manejar la conexión WiFi y MQTT. Se define el pin para el servomotor y el pin para el potenciómetro. Además, se instancia un objeto de la clase Servo para manipular el servomotor.

Se definen las credenciales para la red WiFi y los parámetros del bróker MQTT, como la dirección del bróker, el tema de MQTT y el puerto. Luego, se crean los objetos WiFiClient y PubSubClient, necesarios para establecer la conexión WiFi y la comunicación MQTT, respectivamente.

Luego, se declaran varias funciones que se usarán en el programa: connectToWiFi para manejar la conexión a la red WiFi, connectToMQTTBroker para manejar la conexión al bróker MQTT, mqttCallback para procesar los mensajes entrantes de MQTT, y enviarMensajeMQTT para publicar mensajes en el tema MQTT.

En la función setup, se inicia la comunicación serie a 115200 baudios. Luego, se llama a la función connectToWiFi para conectarse a la red WiFi. Después de eso, se configuran los parámetros del bróker MQTT y se establece la función de callback para manejar los mensajes recibidos. Finalmente, se conecta al bróker MQTT y se adjunta el servomotor al pin especificado.

La función connectToWiFi intenta conectar el ESP8266 a la red WiFi utilizando las credenciales proporcionadas. La función imprime puntos en la consola hasta que la conexión se establece exitosamente, momento en el que imprime un mensaje de éxito.

La función connectToMQTTBroker intenta conectar el ESP8266 al bróker MQTT. Si la conexión falla, se espera cinco segundos antes de intentar de nuevo. Al conectarse exitosamente, se suscribe al tema MQTT definido y publica un mensaje de bienvenida.

La función mqttCallback procesa los mensajes recibidos del bróker MQTT. Si el mensaje es "Alerta", el servomotor gira de 0 a 180 grados y se publica un mensaje indicando que el servomotor ha sido activado. Si el mensaje es "Solucionado", simplemente se imprime un mensaje indicando que el servomotor ha quedado en su lugar original.

En la función loop, se verifica si la conexión al bróker MQTT sigue activa. Si no es así, se vuelve a conectar. Luego,

se llama a mqtt_client.loop para procesar los mensajes entrantes y mantener la conexión MQTT.

Finalmente, la función enviarMensajeMQTT publica un mensaje en el tema MQTT definido. Esta función es llamada dentro del callback cuando se recibe un mensaje de "Alerta", indicando que el servomotor ha sido activado.

IV. Bróker

Para el proyecto se eligió emplear el bróker EMQX, pues se presentaron varios problemas a la hora de tratar de implementar MOSQUITTO.

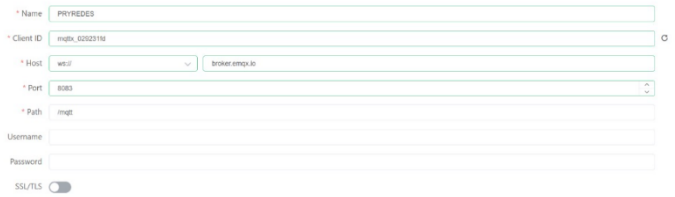

Fig. 10. Configuración de cliente. Elaboración propia.

En la imagen anterior se muestra un formulario para configurar un cliente MQTT que se conectará a un bróker EMQX, donde se da un nombre al cliente, el ID del cliente que es un identificador único para el cliente que debe ser diferente de los IDs de todos los demás clientes conectados al broker, el host o la dirección IP del broker, el puerto predeterminado para MQTT (1883) y la ruta.

Así pues, el cliente se configuró para conectarse al broker EMQX que se ejecuta en el host broker.eng.lo en el puerto 8083 y este se suscribirá al tema /matt. El cliente no está utilizando autenticación y no se está utilizando SSL/TLS.

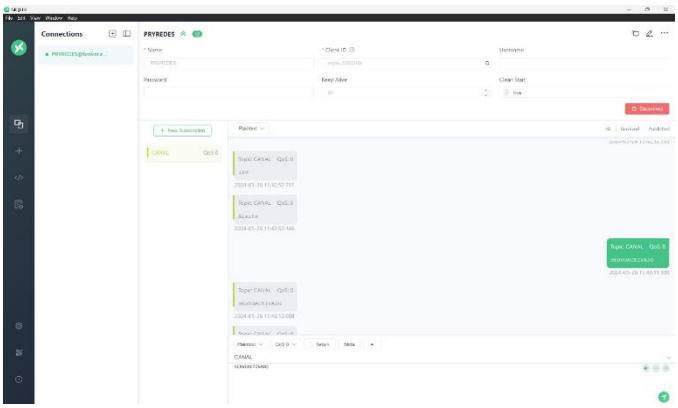

Fig. 11. Muestra de funcionamiento. Elaboración propia.

En esta otra imagen se presenta un ejemplo de su funcionamiento donde cada 5 segundos, como se detallará más adelante, se realiza una lectura del nivel de gas en el tópico canal y se envía una señal que posteriormente se explicará.

V. Aplicación en Java

Para la creación de la aplicación se crearon tres clases: Redes, TelegramBot y Main.

A. *Clase Redes*

```
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class Redes {

    private static final String broker = "tcp://broker.emqx.io:1883";
    private static final String clientId = "JAVA_CLIENT";
    private static final String topic = "CANAL";
    private static final int subQos = 1;
    private static final int pubQos = 1;
    private static final String msg = "Hello MQTT I'm JAVA";

    // Umbral para activar la alerta
    private static final int umbral = 100;

    static long chatId = (long) 6405881095;
    static TelegramBot telegramBot = new TelegramBot();

    public static void run(){
        try {
            MqttClient client = new MqttClient(broker, clientId);
            MqttConnectOptions options = new MqttConnectOptions();
            client.connect(options);
            client.subscribe(topic, subQos);

            if (client.isConnected()) {

                MqttMessage message0 = new MqttMessage(msg.getBytes());
                message0.setQos(pubQos);
                client.publish(topic, message0);

                client.setCallback(new MqttCallback() {
                    public void messageArrived(String topic, MqttMessage message) throws Exception {
                        System.out.println("topic: " + topic);
                        System.out.println("qos: " + message.getQos());
                        System.out.println("message content: " + new String(message.getPayload()));

                        try {
                            // Convertir el mensaje a float
                            float valorFloat = Float.parseFloat(new String(message.getPayload()));
                            System.out.println("Valor como float: " + valorFloat);

                            if (valorFloat > umbral) {
                                String msg1 = "Alerta";
                                MqttMessage message1 = new MqttMessage(msg1.getBytes());
                                message1.setQos(pubQos);
                                client.publish(topic, message1);

                                telegramBot.sendMessage(telegramBot.generateSendMessage(chatId, "*Alerta* Se ha detectado gas metano"));
                            }

                        } catch (NumberFormatException e) {
                            System.err.println("Error al convertir el mensaje a float: " + e.getMessage());
                            // Maneja el error según tus necesidades
                        }

                        String MSG = new String(message.getPayload());
                        System.out.println("MSG: " + MSG);

                        if((MSG.equalsIgnoreCase("SERVOACTIVADO"))==true) {
                            Thread.sleep(5000);
                            String msg2 = "Solucionado";
                            MqttMessage message2 = new MqttMessage(msg2.getBytes());
                            message2.setQos(pubQos);
                            client.publish(topic, message2);

                            telegramBot.sendMessage(telegramBot.generateSendMessage(chatId, "El servomotor ha sido activado correctamente"));
                        }
                    }

                    public void connectionLost(Throwable cause) {
                        System.out.println("connectionLost: " + cause.getMessage());
                    }

                    public void deliveryComplete(IMqttDeliveryToken token) {
                        System.out.println("deliveryComplete: " + token.isComplete());
                    }
                });
            }

        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 12. Código de la clase Redes. Elaboración propia.

Primero, se importan las clases necesarias de la biblioteca Paho MQTT para manejar las conexiones, los mensajes y las callbacks del cliente MQTT. Luego, se define una clase llamada Redes que contiene la lógica principal.

En la clase Redes, se establecen algunas constantes, como la dirección del bróker (broker), el ID del cliente (clientId), el tema al que se suscribirá (topic), los niveles de QoS para la suscripción y la publicación (subQos y pubQos), y un mensaje de prueba (msg). También se define un umbral (umbral) para activar una alerta y se inicializa un ID de chat y una instancia de TelegramBot para enviar mensajes a través de Telegram.

La clase tiene un método estático run, que contiene la lógica principal. Este método intenta crear una instancia de MqttClient utilizando el broker y el ID del cliente. A continuación, se configuran las opciones de conexión (MqttConnectOptions) y se conecta el cliente al broker. Si la conexión es exitosa, el cliente se suscribe al tema especificado.

Se publica un mensaje inicial en el tema para indicar que el cliente se ha conectado. Luego, se establece un callback para manejar los mensajes entrantes, las pérdidas de conexión y la confirmación de entrega de los mensajes publicados.

Dentro del callback, el método messageArrived se ejecuta cuando llega un mensaje al tema suscrito. Este método imprime el tema, el nivel de QoS y el contenido del mensaje recibido.

Intenta convertir el contenido del mensaje a un valor float. Si el valor supera el umbral definido, se publica un mensaje de alerta en el tema y se envía una alerta a través del bot de Telegram.

Si el mensaje recibido es "SERVOACTIVADO", el programa espera 5 segundos antes de publicar un mensaje de "Solucionado" en el tema y envía una notificación correspondiente a través del bot de Telegram.

El método connectionLost imprime un mensaje cuando la conexión se pierde, y el método deliveryComplete imprime un mensaje cuando la entrega de un mensaje publicado se completa.

Finalmente, si ocurre algún MqttException durante la ejecución, se captura y se imprime la traza del error para facilitar la depuración.

B. Clase TelegramBot

```java
import org.telegram.telegrambots.bots.TelegramLongPollingBot;
import org.telegram.telegrambots.meta.api.methods.send.SendMessage;
import org.telegram.telegrambots.meta.api.objects.Update;
import org.telegram.telegrambots.meta.exceptions.TelegramApiException;

public class TelegramBot extends TelegramLongPollingBot {

    @Override
    public String getBotUsername() {
        return "AlarmaMetanoBot";
    }

    @Override
    public String getBotToken() {
        return "7368091474:AAGlgF7QGDLcp6N_ueWT37XYSo2D8tayPnM";
    }

    SendMessage generateSendMessage(Long chatId, String text) {
        SendMessage sendMessage = new SendMessage();
        sendMessage.setChatId(chatId.toString());
        sendMessage.setText(text);
        return sendMessage;
    }

    void sendMessage(SendMessage sendMessage) {
        try {
            execute(sendMessage);
        } catch (TelegramApiException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 13. Código de la clase TelegramBot. Elaboración propia.

Esta clase extiende TelegramLongPollingBot, lo que le permite manejar actualizaciones de Telegram utilizando el método onUpdateReceived, aunque este método no se ha sobrescrito en el código proporcionado.

El método getBotUsername devuelve el nombre del bot, en este caso, "AlarmaMetanoBot". Este nombre es necesario para que el bot se identifique en Telegram.

El método getBotToken devuelve el token de acceso del bot, que es una cadena única proporcionada por BotFather cuando se crea el bot. Este token se utiliza para autenticar las solicitudes enviadas a la API de Telegram.

El método generateSendMessage crea un objeto SendMessage con el ID del chat y el texto del mensaje. Este método toma dos parámetros: chatId, que es el ID del chat al que se enviará el mensaje, y text, que es el contenido del mensaje.

El método sendMessage toma un objeto SendMessage y lo envía utilizando el método execute de TelegramLongPollingBot. Si ocurre una excepción TelegramApiException durante el envío del mensaje, la excepción se captura y su traza se imprime en la consola.

C. Clase Main

```java
import org.telegram.telegrambots.meta.TelegramBotsApi;
import org.telegram.telegrambots.meta.exceptions.TelegramApiException;
import org.telegram.telegrambots.updatesreceivers.DefaultBotSession;

public class Main {
    public static void main(String[] args) {
        try {
            TelegramBotsApi chatBot = new TelegramBotsApi(DefaultBotSession.class);
            chatBot.registerBot(new TelegramBot());
        } catch (TelegramApiException e) {
            e.printStackTrace();
        }
        Redes mqttClient = new Redes();
        mqttClient.run();
    }
}
```

Fig. 14. Código de la clase Main. Elaboración propia.

Esta última clase, primero se intenta crear una instancia de TelegramBotsApi, especificando que debe usar la clase DefaultBotSession. Este objeto es responsable de manejar las sesiones de bot y la comunicación con la API de Telegram.

Dentro del bloque try, se registra el bot de Telegram utilizando el método registerBot, pasando una nueva instancia de la clase TelegramBot. Esto hace que el bot esté listo para recibir y enviar mensajes a través de Telegram.

Si ocurre alguna excepción de tipo TelegramApiException durante la configuración o el registro del bot, esta se captura y se imprime la traza del error en la consola, facilitando así la depuración de problemas relacionados con la API de Telegram.

Después de configurar el bot de Telegram, se crea una instancia de la clase Redes, que representa el cliente MQTT. Se llama al método run de esta instancia para iniciar la conexión al broker MQTT y gestionar la lógica de suscripción y publicación de mensajes.

VI. BOT DE TELEGRAM

Para Para programar un bot en Telegram, primero se creó una cuenta activa en Telegram. Una vez que se tuvo la cuenta, se abrió la aplicación y se buscó el bot llamado BotFather, que es el administrador oficial de bots en Telegram. Se inició una conversación con BotFather y se utilizó el comando /newbot para crear un nuevo bot. BotFather pidió proporcionar un nombre y un nombre de usuario único que terminara en "bot".

Una vez completado este paso, BotFather proporcionó un token de autenticación, crucial para interactuar con la API de Telegram, como se presenta en la Figura 13.

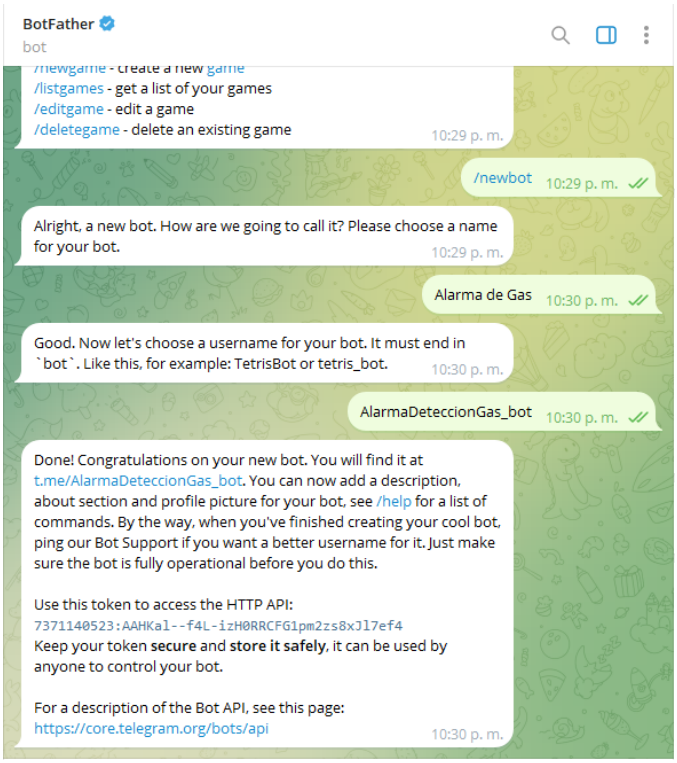

Fig. 15. Creación de Bot. Elaboración propia.

Con el token en mano, el siguiente paso fue configurar el entorno de desarrollo. Se decidió usar Spring Boot para crear el bot y se comenzó creando un nuevo proyecto de Spring Boot. Se incluyó la dependencia específica para Telegram en el archivo pom.xml, ya que se usó Maven. Esta dependencia se encontró en el repositorio central de Maven y se añadió correctamente para garantizar que el proyecto pudiera comunicarse con la API de Telegram.

Fig. 16. Pom. Elaboración propia.

Dentro del proyecto de Spring Boot, se creó la clase TelegramBot que, como se mencionó previamente, extendió una de las clases proporcionadas por la biblioteca de Telegram que se había añadido como dependencia. En esta clase, se sobrescribieron algunos métodos esenciales, como los que manejan los mensajes entrantes para que en lugar de responder a los mensajes de un usuario empleara la subscripción a tópicos. Se configuró correctamente el token de autenticación obtenido de BotFather dentro de esta clase para que el bot pudiera comunicarse de manera efectiva con la API de Telegram.

Una vez configurada la clase principal del bot, se implementó la lógica que definió cómo el bot respondería a los mensajes y se incluyó su activación en la clase Main.

VII. Diagramas

A continuación, se puede ver el diagrama de clases en la figura 17.

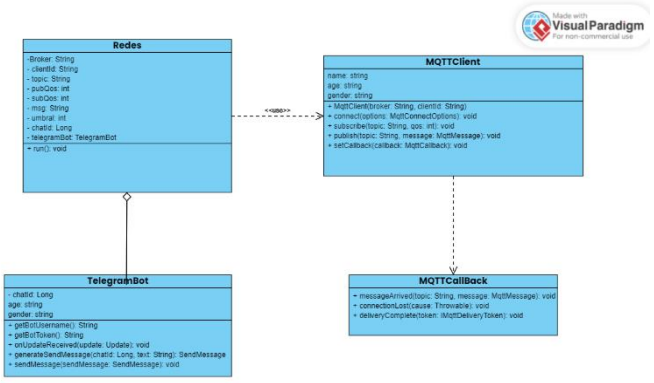

Fig. 17. Diagrama de clases. Elaboración propia.

En el diagrama se pueden observar los atributos y métodos que se usa en cada una de las clases involucradas en el sistema.

Como siguiente paso se realizó un diagrama de secuencia para plasmar el funcionamiento global del sistema. Observar figura 18

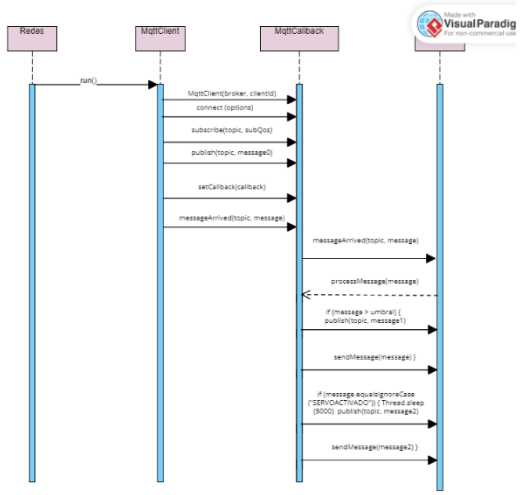

Fig. 18. Diagrama de secuencia .Elaboración propia.

Como se puede observar en la figura, el proceso comienza con la inicialización y ejecución de la clase Redes, la cual llama

al método run() de mqttClient. Posteriormente, este cliente mqtt se inicializa con el bróker y el clienId, y se suscribe al tema utilizando suscribe (topic, subQos). Luego, publica un mensaje de prueba y configura el callback.

Siguiendo con lo antes mencionado, cuando el mqttClient recibe un mensaje, este invoca al método messageArrived(topic, message) y analiza si el mensaje supera un umbral predefinido anteriormente. En caso de que se supere el umbral predefinido, se publica un mensaje de alerta y se envía a través de telegramBot. Si el mensaje recibido es "ServoActivado", el callback activa el servomotor y espera un tiempo predeterminado. En este punto se publica un mensaje de resolución y se envía una confirmación a través de TelegramBot.

VIII. Pruebas Finales

Como punto final, se realizaron unas pruebas para verificar el funcionamiento de los dispositivos y del sistema. La primera prueba consistía en acercar el dispositivo a una estufa de gas con el filtro abierto. El objetivo de esta prueba era comprobar si el dispositivo lograba detectar correctamente los niveles de gas altos y enviar la alarma. En la prueba se observó que luego de transcurrido un tiempo, los sensores fueron capaces de detectar el nivel de gas por encima del umbral determinado anteriormente, enviando la alerta y activando el servomotor. Posteriormente, se prende el led rojo (indicando alerta) y se activa el buzzer (observar figura 19). Una vez solucionado el problema se prende el led color verde y se envía el mensaje al bot de telegram informado que el servomotor fue activado correctamente (observar figura 20).

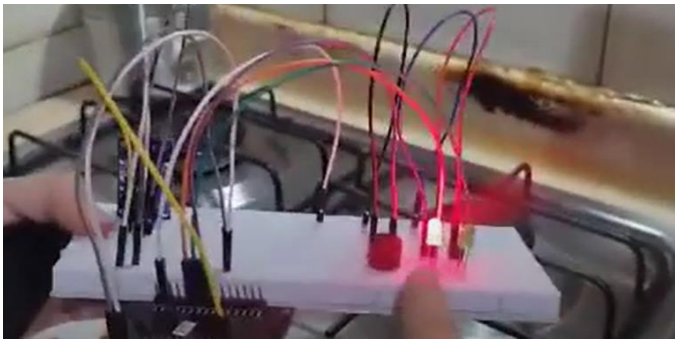

Fig. 19. Activación de led y buzzer .Elaboración propia.

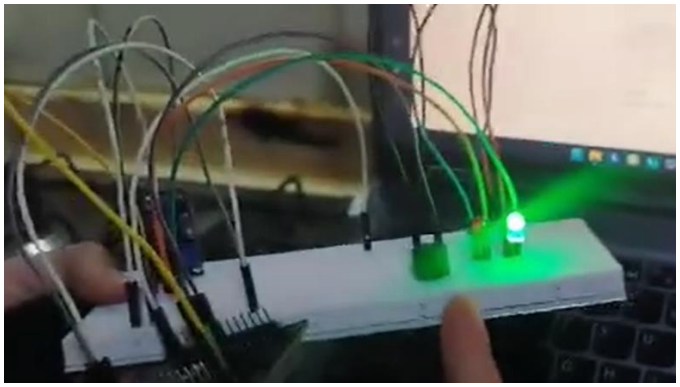

Fig. 19. Activación de led verde al solucionar el problema .Elaboración propia.

Esta prueba cumple con los objetivos planteados ya que se logra verificar el correcto funcionamiento de los dispositivos y el sistema.

La segunda prueba consistía en realizar el mismo proceso pero ahora con un mechero, y al igual que con la prueba anterior, su objetivo era verificar el correcto funcionamiento de los dispositivos y el sistema. Sin embargo, al realizar el proceso se encontró que la prueba del mechero no es una prueba efectiva dado que este tiene un gas distinto a los que el MQ2 es capaz de detectar.

Para visualizar mejor la primera prueba, visualice el video del siguiente link: https://youtu.be/HRxF4fIEAgU

Referencias

[1] WHO. Global health risk: mortality and burden of disease attributable to select major risk. Geneva: WHO; 2009, https://iris.who.int/bitstream/handle/10665/44203/9789241563871_eng.pdf?sequence=1

[2] World Health Organization. The World Health Report 2002: Reducing risks, Promoting Healthy Life. Geneva, Switzerland: World Health Organization; 2002.

[3] L. C. Fernández, "Contaminación del Aire interior y su impacto en la Patología Respiratoria," Archivos de Bronconeumología, https://www.sciencedirect.com/science/article/abs/pii/S0300289612001196?fr=RR-7 (accessed May 1, 2024).

[4] Hunkeler, U., Truong, H. and Stanford, A. (no date) *Mqtt-S — a publish/subscribe protocol for Wireless Sensor Networks | IEEE conference publication | IEEE xplore*. Available at: https://ieeexplore.ieee.org/document/4554519/ (Accessed: 29 April 2024).

[5] Soni, D. and Makwana, A. (2017) *(PDF) A survey on MQTT: A protocol of internet of things(iot)*. Available at: https://www.researchgate.net/publication/316018571_A_SURVEY_ON_MQTT_A_PROTOCOL_OF_INTERNET_OF_THINGSIOT (Accessed: 29 April 2024).

[6] HiveMQ (2020) *What is MQTT | Mqtt Essentials Part 1, YouTube*. Available at: https://www.youtube.com/watch?app=desktop&v=jTeJxQFD8Ak (Accessed: 30 April 2024).

[7] HiveMQ (2020a) *Pub sub model | MQTT essentials part 3, YouTube*. Available at: https://www.youtube.com/watch?v=HCzQJMdHcy0 (Accessed: 30 April 2024).

[8] *Eclipse mosquitto* (2018) *Eclipse Mosquitto*. Available at: https://mosquitto.org/ (Accessed: 01 May 2024).

[9] Automation Station (2020) *How to configure an MQTT Mosquitto broker and enable user authentication on Windows, YouTube*. Available at: https://www.youtube.com/watch?app=desktop&v=72u6gIkeqUc (Accessed: 01 May 2024).

[10] 1 Ud./6 Uds. ESP8266 NodeMCU LUA CH340 ESP-12E Placa De Desarrollo De Internet WiFi 157.48in, Temu, NodeMCU ESP8266 (accessed May 1, 2024).

[11] "Servomotor SG90 Azul electrónica proyectos Arduino Robótica - $ 9.900," Amazon, Servomotor (accessed May 1, 2024).

[12] "Protoboard Mediana 400 puntos (8.5x5.5 cm) - $ 9.630," Mercado Libre, Protoboard (accessed May 1, 2024).

[13] Vistronica, "Modulo sensor de detección de gas MQ-2," VISTRONICA S.A.S, MQ-2 (accessed May 1, 2024).

[14] "Resistencia 4.7K Ohm 1/4w 1% X5 Unidades," Arca Electrónica, Resistencia (accessed May 1, 2024).

[15] "Cables jumpers macho macho 20 cm x 40 unidades - $ 12.000," Mercado Libre, Cable Macho Macho (accessed May 1, 2024).

[16] "Pila Batería Cuadrada Carbón zinc 9v juguete multímetro x5 - $ 12.880," Mercado Libre, Bateria 9V (accessed May 1, 2024).

[17] "Cable Micro USB De Carga Rápida De 5A Cable De Datos Micro USB," Temu, Cable Micro USB (accessed May 1, 2024).

[18] "Modulo buzzer (FC-49)," Moviltronics, Buzzer (accessed May 1, 2024).